Embedded Systems Code Compression
Group 243 Spring 2006

Final Report

Andrew Aarestad
Joseph Thomas
Kyaw Min

Embedded Systems Code Compression
Final Report Group 243

Table of Contents

Andrew Aarestad
Joseph Thomas
Kyaw Min

Documents Included:

1 - Dictionary-Based Instruction Compression and the TMS320C6000 Architecture

A white paper describing the compression schemes we developed and the results of the tests we did to evaluate their performance

7 - Software Description Supplement

Since we did not discuss our software at length in the white paper, we have included a supplement that describes our code in depth.

10 - Budget

A full list of all expenses incurred during this project

11 - Project Comments

Our reactions to the project

12 - User's Guide

A tutorial on how to use the code we have written and used

15 - Appendices

A hard copy of our source code is attached.

Dictionary-Based Instruction Compression and the TMS320C6000 Architecture

Andrew Aarestad
Joseph Thomas
Kyaw Min

1. Introduction

As solid-state memory technology develops and becomes cheaper, design parameters for devices that use it can change as well. For years, one of the primary goals of computer engineering, especially in the embedded systems field, was the reduction of the memory footprint requirements of software. Now that memory is becoming increasingly larger and more affordable, the design focus is shifting away from footprint reduction to reduction of system power consumption. Power consumption has always been a concern for computer engineers, but only recently has it moved to the forefront of engineering design metrics. New methods of data storage are being developed that not only reduce memory footprint requirements but also reduce memory bus traffic, a major contributor to the overall system power consumption [1]. Instruction compression is one of the techniques being applied to reduce power consumption.

In this paper, we present several schemes that we have developed for use in the Texas Instruments TMS320C6x VLIW architecture. We explore the various performance tradeoffs of modifications to the original architecture, adding compression to the scheme in several places.

2. Design Parameters

Energy Savings

The primary design goal of the compression system is to reduce the energy requirements of the system. This is achieved by increasing the cache hit rate. Fewer cache misses means the CPU has to go off-chip to satisfy its instruction requests less. However, we do not provide an analysis of the gains in terms of power savings since the power requirements of the memory system varies by implementation. Instead, we provide an analysis of the hit rate gains which, given information about the power consumption of the memory bus, could be converted to data about the power consumption.

Transparency

One of the design goals for this project is transparency to the CPU core. This means that one of our requirements is that modifications to the core are unnecessary and unwanted. Our modifications include positioning a decompression unit between different levels of instruction cache. This does mean modification of the chip since the cache is on-chip, but does not require modification of the core. The idea is to allow the core to behave exactly as if there were no modifications. This is an important feature when designing for processors with proprietary cores such as the one we are using.

Cache Bus Widths

The original system as defined in TI's documentation calls for L1 and L2 caches line sizes of 512 and 1024 bits, respectively [2]. The TMS is a VLIW processor, which means that it processes a fetch packet of up to eight instructions per clock cycle. Each instruction is 32 bits, so the cache lines for the L1 and L2 caches represent 2 and 4 fetch packets, respectively. To implement the compression schemes we have proposed, redesign of the cache busses would be required since fewer bits would need to be transferred to satisfy the same requests.

Chip Cost

We have attempted to design our compression schemes in such a way that the cost of the chip does not increase. Since we have allocated cache-speed memory for use in decompression, we have 'borrowed' the memory from the L2 cache. Thus, in our analysis we have reduced the size of the L2 cache to keep the total memory requirements of the chip constant.

3. Compression Schemes

Both of these modifications of the TI TMS320C6x architecture compress machine instructions by placing an instruction decompression unit on the chip (Fig 1). Instructions are compressed before they are placed in main memory from 32 to 16 bits using a dictionary compression algorithm similar to those seen in [3] and [4]. The compressed version of the program and a decompression dictionary are loaded into the DSP's memory. The decompression unit has a small amount of local memory that is used to hold the dictionary. When the CPU requests a 256-bit fetch packet located at a certain address, the decompression unit intercepts the request, translates the address into the compressed address space, fetches the compressed instructions, and decompresses them before returning them to the CPU. The difference between the two schemes we propose here is the location of the decompression unit in the cache system.

The following descriptions of what happens during an instruction fetch illustrate our changes to the original system and give a side-by-side comparison of the actions of both our schemes.

Scheme 1

1. Address is generated in CPU core

2. Address is sent to L1 cache. As in the original configuration, the L1 cache contains uncompressed fetch packets. If the requested fetch packet is not located in the L1 cache, the cache updates.

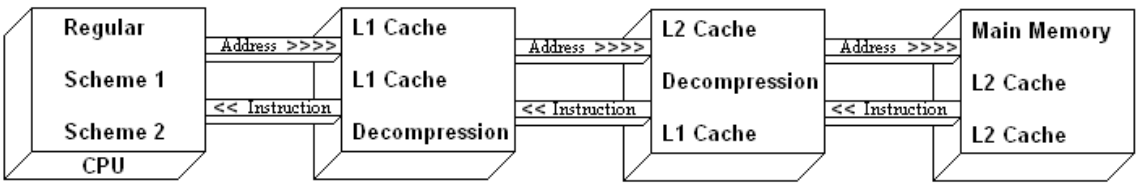

Figure 1 - Shows the placement of the decompression unit for the schemes.

3. Fetch packet is returned if there is a L1 cache hit, otherwise the address is sent to the decompression unit.
4. Address is translated into compressed address space. Since the com pressed instructions are half the size of the original, the address of a compressed instruction will be half of the original request.
5. Address is sent to the L2 cache.
6. The L2 cache is searched for the compressed address.
7. Compressed cache line is returned to decompression unit if there is a L2 hit, otherwise the requested address is sent to the DMA unit.
8. Requested instructions are fetched from main memory.
9. L2 cache line is received from main memory, L2 cache updated.
10. Compressed cache line is sent to decompression unit.
11. Cache line is decompressed.
12. Decompressed cache line is sent to the L1 cache.
13. L1 cache is updated.
14. Decompressed fetch packet is sent to CPU.

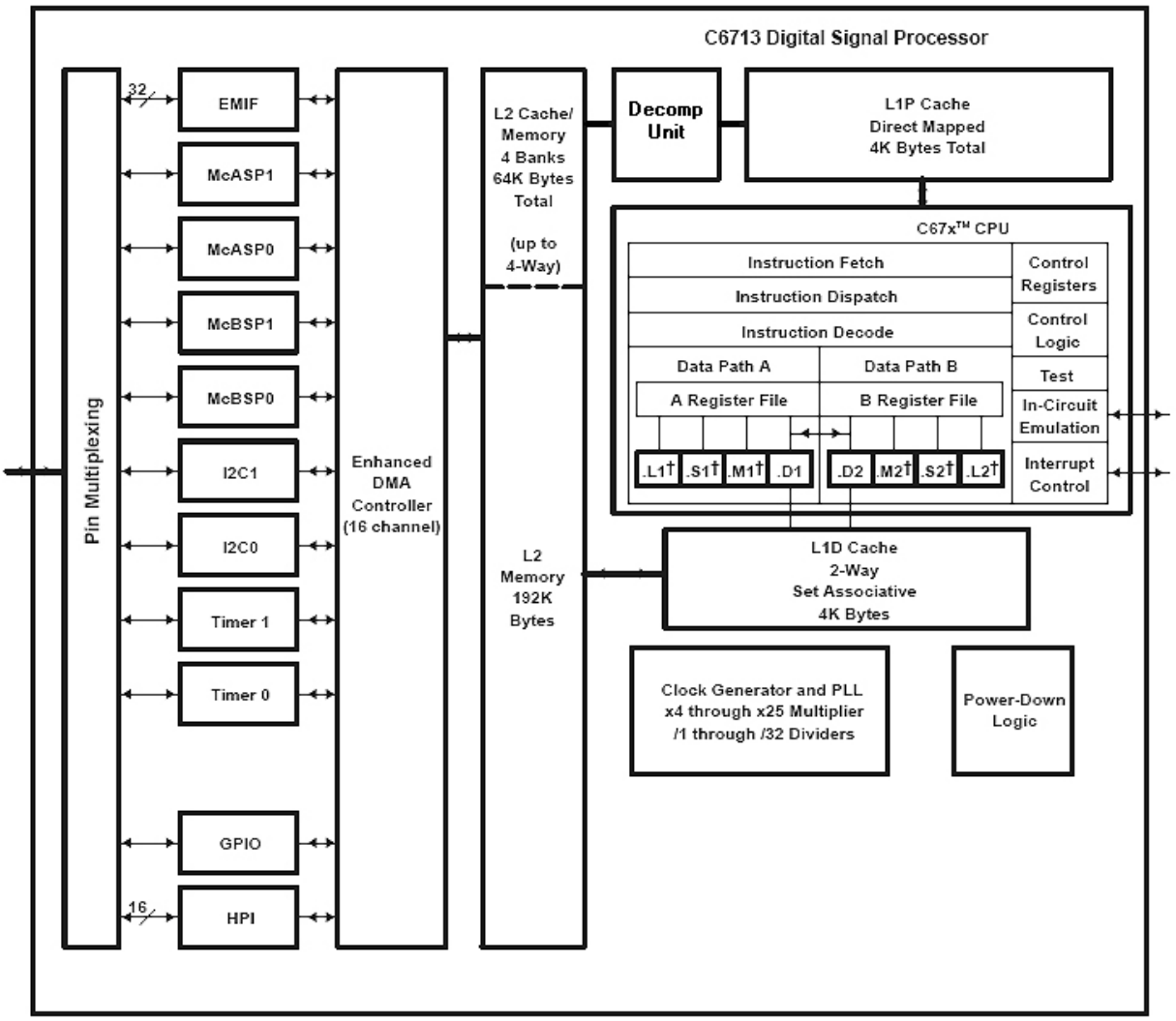

Figure 2 - The TMS320C6713 architecture [5] with decompression unit inserted for scheme 1.

Scheme 2

1. Address is generated in CPU core
2. Address is sent to decompression unit
3. Address is translated into compressed address space
4. Compressed address is sent to L1 cache
5. L1 is searched, if it misses, the address is sent to the L2 cache
6. L2 is searched, if it misses, it is updated from main memory
7. Instructions are fetched from main memory
8. Instructions are received, L2 frame is updated
9. Compressed cache line is sent to L1
10. L1 cache line is updated
11. Compressed fetch packet is sent to decompression unit
12. Fetch packet is decompressed
13. Decompressed fetch packet is sent to the CPU

4. Performance Analysis

We have evaluated the performance of our compression schemes using the following metrics:

- Memory Footprint Reduction
- Change in Cache Hit Rate
- Instruction Access Time Penalty

Memory Footprint

Both schemes use static-length 16-bit codewords. This gives us an even 50% reduction of the program size, but the file must also contain the decompression dictionary. In smaller programs, this may actually result in code expansion. See Fig. 2 for data relating to memory footprint performance of our schemes.

Cache Hit Rate

The primary goal of our compression was to increase the effective cache hit rate. By compressing the caches, we are able to store more fetch packets in cache for the same cache size. With more chances for a cache hit, there will be fewer cache misses.

Access Time Penalty

The tradeoff that comes with the cache hit rate increase is a penalty in instruction access time. A compressed fetch packet is half as long as an uncompressed one. Because the memory controller must bring the instructions of the fetch packet into the chip in serial, only half as many of these fetches would occur per fetch packet, reducing the L2 miss penalty significantly. However, the penalty is increased by the delay caused by decompression. For a basis for the ratio of the access times for different levels of cache, we consulted [6].

5. Conclusions

To analyze the performance of our schemes, we simulated several DSP-specific applications on Vinodh Cuppu's c6xsim cycle-accurate simulator [7]. We then ran the instruction trace generated through a cache performance simulator that we developed. This simulator is more thoroughly documented in our document "TMSCache Cache Performance Simulator User's Guide." [REF]

Scheme 1 has the same L1 cache performance as the original system since it leaves the L1 cache uncompressed. Gains are noticed in the L2 cache hit rate, see Fig. 3 for data. Since the L2 cache is updated serially, the L2 penalty is reduced by compression. This penalty reduction compensates for the penalty increase caused by decompression and produces an overall decrease in average instruction access time. See Fig. 4 for data about the average access times.

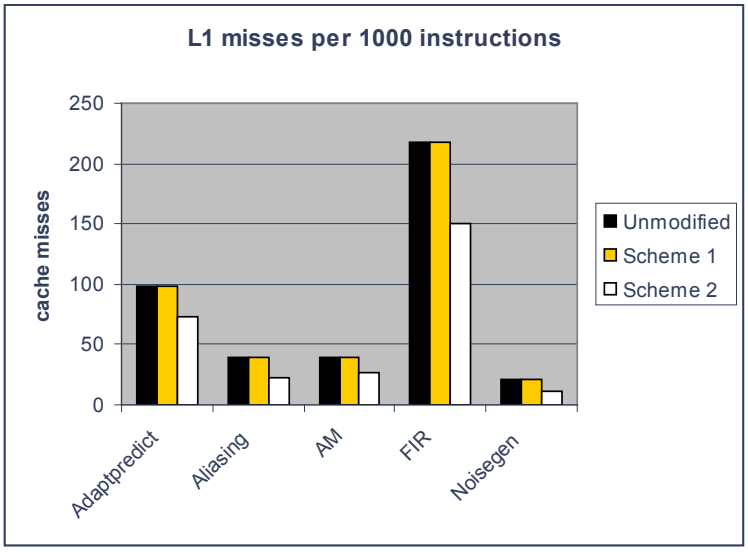

Figure 3 - L1 cache misses

Scheme 2 had better cache miss rates than Scheme 1 or the original, but does so at the cost of access time. Since it must decompress every instruction issued, the decompression penalty adds up fast. As is illustrated in Fig. 3, the average access time is increased by about a factor of three. For a non-time-critical system in which energy savings is paramount, a system such as this could be desirable, but for most systems this is too large of a penalty.

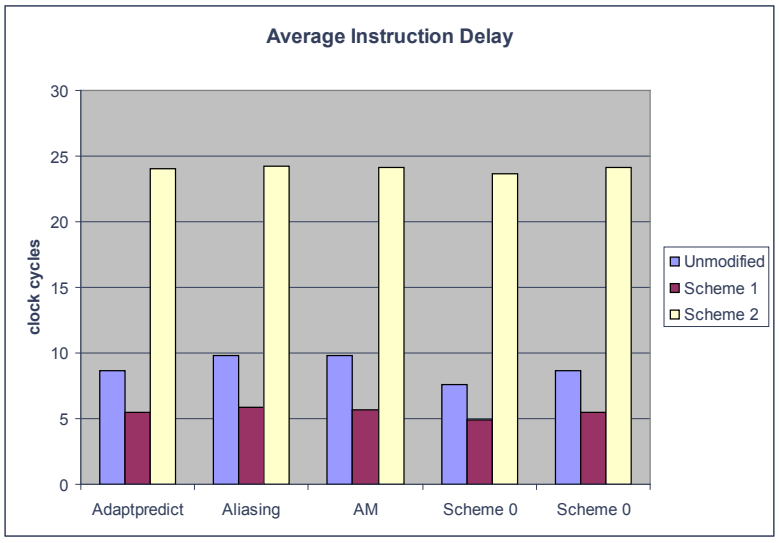

Figure 4 - Average instruction delay for our test programs

In conclusion, our data shows that implementing a compression technique such as the ones we illustrated here can result in both energy savings and performance gains without increasing the cost of the chip.

References

[1] Luca Benini, Alberto Macii, and Enrico Macii, " Minimizing Memory Access Energy in Embedded Systems by Selective Instruction Compression," IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 10, No. 5, pp. 521-530, Oct 2002.

[2] TMS320C6713 Floating-Point Digital Signal Processor: Texas-Instruments, 2005.

[3] Luca Benini, Francesco Menichelli, and Mauro Olivieri, "A Class of Code Compression Schemes for Reducing Power Consumption in Embedded Microprocessor Systems," IEEE Transactions on Computers, vol. 53, No. 4, pp. 467-482. Apr. 2004.

[4] Montserrat Ros, and Peter Sutton. "Code Compression Based on Operand-Factorization for VLIW Processors," IEEE Proceedings of the Data Compression Conference, pp. 1-5, Apr 14, 2004.

[5] TMS320C6000 CPU and Instruction Set Reference Guide: Texas-Instruments.

[6] J. L. Hennessy and D.A. Patterson, Computer Architecture – A Quantitative Approach, 3rd ed. San Mateo, CA: Morgan Kaufmann, 1996.

[7] Vinodh Cuppu, Cycle Accurate Simulator for TMS320C62x, 8 way VLIW DSP Processor: University of Maryland, College Park, 1999.

Embedded Systems Code Compression
Final Report Group 243

Software Description Supplement

Andrew Aarestad
Joseph Thomas
Kyaw Min

Our Software

1. CodeScan

Codescan is a 32 to 16 bit dictionary based code compression program. This means that it takes in instructions and compresses them using indexes into a lookup table.

The first part of Codescan is reading in the data. See Fig. 1 for a flowchart. Codescan reads in the disassembly output of CCStudio, which contains all of the 32 bit opcodes for a program to be placed onto the DSP. The program assigns each unique opcode a space in a dictionary array. Next, it goes back through the code again and replaces each instruction will its index in the dictionary array. We achieve compression because we can assign the original 32 bit opcode a 16 bit index. That is a 50% reduction. This happens to every opcode in the program. Finally, the program adds a flag after the compressed instructions and appends the dictionary to the end. The flag allows us to determine the end of the compressed program and the beginning of the dictionary. We then send all this information to an output file.

The compression is a nice byproduct of our main goal, which was to focus on reducing the energy requirements of the system. The compression will never be 50% because we have to add the dictionary to the end to correctly reassemble the instructions. Compression of about 90% is typical. Also, there is a possibility that code expansion will occur due to the addition of the dictionary. This happens when there are too many unique instructions, as often is the case in small programs. In the worst case with all unique instructions, a 150% compression ratio will result.

We are able to compress any set of 32 bit opcode printouts without any problems. The reduction of memory footprint is a plus, but the major use of the compression is to allow for more information to be stored in cache near the processor. Creating a larger effective cache combined with a smaller memory footprint is what Codescan was created to do.

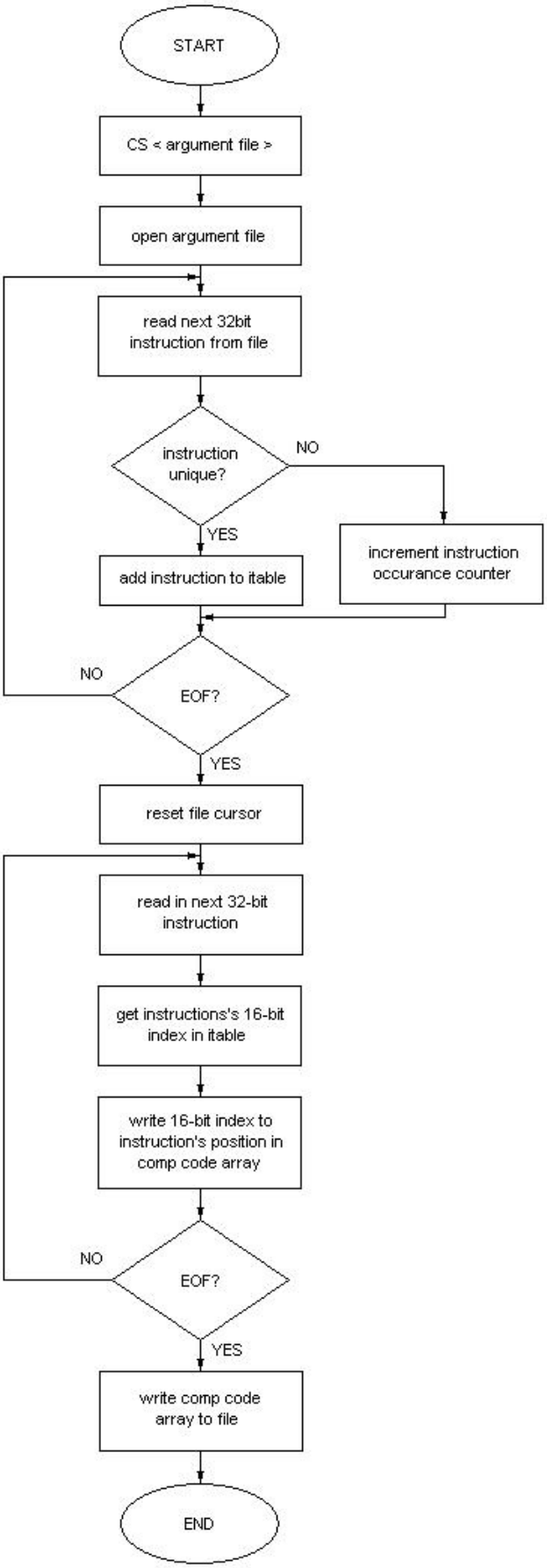

Figure 1 - Flowchart for CodeScan algorithm

2. TMSCache

This program is a cache performance simulator that takes a program trace and generates cache performance data for our different compression schemes. For a full description of the data generation process, please refer to the User's Guide.

TMSCache processes an instruction trace and keeps track of the cache hits/misses. It also calculates the estimated average instruction delay in terms of clock cycles. It calculates these data for the original architecture of the TMSC6x, for our Scheme 1, and for Scheme 2. For a description of our compression schemes, see "Code Compression and the TMSC6000 Architecture."

TMSCache was written by us in C. It uses some GNU/Linux extensions, and thus must be compiled with an appropriate compiler. It also requires the c6xsim package, described below.

Other Software

3. C6xsim

Written by Vinodh Cuppu, c6xsim is a cycle-accurate simulator of the TMSC6000 architecture. We use it to generate the instruction traces used by TMSCache. For a full description of this process, please refer to the User's Guide.

We have modified c6xsim to suit our needs. Originally it was not configured to produce an instruction trace, so we modified the code to produce an output file called tracefile.txt which contains the trace.

Embedded Systems Code Compression
Final Report Group 243

Budget

Andrew Aarestad
Joseph Thomas
Kyaw Min

Supplies Required/Purchased

1.	CCStudio/DSP	Free - Available for our use in ECE 237
2.	Linux Development Environment	Free - Available through NDSU SOD cluster
3.	Textbook: Embedded Computing, First Edition: A VLIW Approach to Architecture, Compilers and Tools	$75 from Amazon.com

Embedded Systems Code Compression
Final Report Group 243

Project Comments

Andrew Aarestad
Joseph Thomas
Kyaw Min

Our Reactions

Project Issues

A major issue with our project was the inability to use CCStudio to its fullest potential. We know it probably has the ability to pretty much everything we want it to do, but it is a very picky and unforgiving program. We also found it difficult to find specific information for the processor we were using. Texas Instruments does not publish much of the information we wanted. We eventually found most of what we needed, but it required a lot of time dedicated to research early in the project.

Lessons Learned

Research is a very difficult thing. We didn't think that it would be so difficult and time consuming. After we finally found what we needed to read and finally understood it, we then needed to apply it meaningfully. We were accustomed to the linear, predictable style of most undergraduate work, so this project was a new experience for all of us.

Future Improvements

The next step in this project would be the integration of our compression into a simulator such as the c6xsim simulator. This would be quite an involved process and would probably require more faculty guidance than our project due to the level of detail in a processor simulator. However, should a group want to attempt this task, we have laid out a useful base for them to start from. The next group would have to figure out how the coff .out file works. After 2 semester we still don't fully understand how that works.

There is also room for improvement in the compression algorithm. The technique we use is fairly basic, so by using a more complex compression technique, further gains could be acheived.

TMSCache Cache Performance Simulator User's Guide

Andrew Aarestad
Joseph Thomas
Kyaw Min

This document does not describe the theory behind the compression system the following software is designed to test. For a complete description of the compression theory, please refer to our document "Dictionary-Based Instruction Compression and the TMS320C6000 Architecture."

Table of Contents

1. Introduction
2. Package Contents
3. Building Executables
4. Creating COFF Files
5. Generating Trace Files
6. Obtaining Performance Data
7. Troubleshooting

1. Introduction

This software (TMSCache) was written for use in conjuction with c6xsim, a cycle-accurate simulator of a TMSC6000 series processor. It is assumed that the user has access to a compiler for the TMS such as CCStudio. CCStudio requires Windows, but c6xsim and TMSCache require a GNU/Linux environment. While it may be possible to compile the sources provided in Windows, it is recommended that you compile and execute c6xsim and TMSCache using Linux. The easiest way to do this in combination with using CCStudio is to have access to two computers, and to transfer the files produced by CCStudio to the Linux machine. This tutorial will assume that user has access to two machines and a basic knowledge of the two operating systems.

Follow these steps to produce cache performance data with TMSCache:

- Compile the provided source files into executables.
- Compile a test program using CCStudio or any compatible compiler.
- Simulate the COFF file using c6xsim.
- Run TMSCache using the trace file produced by c6xsim as input.

2. Package Contents

- tmscache.tar.gz - source files
- User's Guide - This document
- testcoff.out - sample coff file

3. Building Executables

Start by extracting the archive to its own directory:

```
$ mkdir tmscache
$ gunzip tmscache.gz ./tmscache
$ cd tmscache
```

Compile the tmscache and c6xsim sources as follows:

```
$ gcc -Wall -o tmscache tmscache.c
$ make
```

The directory should now contain the executable files for both tmscache and the c6xsim simulator. This will also be the directory where you will copy the coff files generated in the next section.

4. Creating COFF Files

Once the two pieces of software are ready, it is time to compile a test program. Use a benchmark suite such as the mediabench collection to obtain a wide variety of application-specific data. This guide will not cover the use of CCStudio, except to say that once the program has been compiled, CCStudio will create a coff file with extension .out that is to be used in the simulator. Copy this file from the Windows machine you compiled it on to the tmscache directory on the Linux machine.

A sample coff file, testcoff.out, has been included in the package and will be used as an example.

5. Generating Trace Files

Once the coff file has been generated, run the program on the c6xsim simulator:

```
$ c6000 testcoff.out
```

You should see the simulator begin to display dots signifying that it is simulating the program, and every so often it will give a performance report. C6xsim has been modified to produce an instruction trace that is to be used as the input to tmscache. By default it will trace 100,000 instructions, but this can be changed by modifying the global TRACE_LEN in c6000.c. After the trace is complete, it will create an ASCII text file of the instruction trace with one instruction per line.

You are now ready to generate cache performance data with tmscache.

6. Obtaining Performance Data

After c6xsim has completed, run the trace file through tmscache:

```
$ tmscache tracefile.txt
```

TMSCache will display performance data, along with generating a data file. Also, various debug information can be displayed/dumped by turning on different options:

```
syntax: tmsim -<options> <filename>

options:    v1 - semi-verbose instruction trace
            v2 - full debug info
            d1 - dump small debug file
            d2 - dump all cache info *LARGE FILE*

example: tmsim -v1d2 trace1.txt
```

7. Troubleshooting

C6xsim was written for an older version of the CCStudio compiler and commonly produces the following errors:

- The simulator does not recognize the file. Make sure the coff file is in the same directory as c6xsim, referencing a file in another directory may produce this error.

```
[coffload.c:prog_load, line 47] unable to open target
binary `coffs/testcoff.out' for getting file size, er-
rno 2
```

- The simulator executes for a while, and then produces an error similar to this:

```
unknown instruction with opcode 0x3c in .S unit in E1
pipeline stage @ 1571 cycle, instruction is <x>, pc <y>
```

In this case, the simulator is usually still able to run long enough to generate a sufficiently long instruction trace. Perhaps using an older version of CCStudio to generate the coff files would prevent this error.

Embedded Systems Code Compression
Final Report Group 243

Appendix

Andrew Aarestad
Joseph Thomas
Kyaw Min

Source code for TMSCache.c:

```
////////////////////////////////////////////////////////////////////////////
//
//  TMSCache TMS320C6000 Cache Performance Simulator
//
//  This is a cache simulator for the Texas Instruments TMS320C6000 series of DSPs.
//  It does not actually execute any code - it runs through an instruction trace
//  that is to be generated using the c6xsim processor simulator.
//
//  We collect data about the instruction cache performance. We wrote this program
//  to test the compression schemes that we designed as modifications of this
//  architecture. This code generates cache hit/miss data using three schemes:
//
//             Scheme 0 - No Compression, Original Cache Structure
//             Scheme 1 - L2 Cache is compressed, L1 remains uncompressed
//             Scheme 2 - L1 Cache is also compressed.
//
//  We collect the following data about cache performance:
//
//      - Cache misses in 1st 1000 instructions (ie starting with an empty cache)
//      - Cache misses per 1000 instructions after X instructions already occur (ie starting data
collection with cache full)
//             - Average # of CPU stall cycles per instruction
//
////////////////////////////////////////////////////////////////////////////

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include <sys/types.h>

#define S0_L1_SIZE 16                // Number of Frames in S0 L1 cache
#define S0_L2_SIZE 128               // Number of Frames in S0 L2 cache
#define S1_L1_SIZE 16                // Number of Frames in S1 L1 cache
#define S1_L2_SIZE 224               // Number of Frames in S1 L2 cache
#define S2_L1_SIZE 32                // Number of Frames in S2 L1 cache
#define S2_L2_SIZE 512               // Number of Frames in S2 L2 cache

#define L1_HITTIME 1                 // CPU cycles stalled for L1 hit
#define L2_HITTIME 20                // CPU cycles stalled for L2 hit
#define L2_PENALTY_COMP 200          // CPU cycles stalled to access compressed main memory
#define L2_PENALTY_UNCOMP 400 // CPU cycles stalled to access uncompressed main memory
#define DECOMP_PENALTY 20            // CPU cycles stalled to decompress a FP
```

```
#define VERSION_NUM 4.22

//////////////////////////////////////////////////////////////////////////////
// Data Structures
//////////////////////////////////////////////////////////////////////////////

struct frame
{
       int address;                         // Address of FP
    unsigned int lastused;           // When an L2 frame is updated, the current instnum is stored in
lastused for LRU bookkeeping
};

struct performance_data
{
       int L1_misses;                       // Total number of L1 cache misses
       int L2_misses;                       // Total number of L2 cache misses
       int L1_misses_2nd1000;       // Number of L1 cache misses during the 2nd 1000 instructions
       int L2_misses_2nd1000;       // Number of L2 cache misses during the 2nd 1000 instructions
       int *accesstimes;                    // Array containing access times of each instruction
       float avgaccesstime;         // Average instruction access time, calculated after simulation
};

struct fdata
{
       FILE *infile;                        // file pointer for input file
       FILE *debugfile;                     // file pointer used in debug dumping
       FILE *datafile;                            // file pointer used for dumping final data
       char *infilename;                    // string containing name of input file
       int *trace;                                // Instruction trace
       int instcount;                       // number of entries in the input trace file

} file_data;

struct sim_status
{
       char *name;                                // Initialization call specifies an identifier
string
       int L1comp;                                // Flag designating compression in the L1 cache
       int L2comp;                                // Flag designating compression in the L2 cache
       int L1size;                                // Number of entries in Scheme's L1 cache
       int L2size;                                // Number of entries in Scheme's L2 cache
       int instnum;                         // Total number of instructions processed, used in LRU
replacement for L2 cache
       int PC;                                    // Address of current instruction being processed
       struct frame *L1, *L2;       // caches get malloc'ed in init_status
};

//////////////////////////////////////////////////////////////////////////////
// Global Debug Variables
//////////////////////////////////////////////////////////////////////////////

int verbose = 0;                            // Limits output to a brief summary of each instruction
being processed
int full_debug = 0;                         // Full instruction debugging info
int debug_fulldump = 0;                     // outputs full debug info to file (warning - this is a
lot of info)
int debug_smalldump = 0;             // Outputs some debug info to file

//////////////////////////////////////////////////////////////////////////////
// Functions
//////////////////////////////////////////////////////////////////////////////

void printsyntax();
void read_file(char *arg1);
void runtrace(struct sim_status *current_status,struct performance_data *pdata);
int searchL1(struct sim_status *current_status,int addr);
int searchL2(struct sim_status *current_status,int addr);
int updateL1(struct sim_status *current_status,int addr,struct performance_data *pdata);
int updateL2(struct sim_status *current_status,int addr,int *source);
```

```
void init_status(char *scheme,struct sim_status *current_status,int size_L1,int size_L2,int
L1comp,int L2comp);
void init_pdata(struct performance_data *pdata);
void dumpcache(struct frame *cache,int size,int replaced_index);
void printresults(struct performance_data *pdata);
void dumpdata(struct performance_data *pdata);

////////////////////////////////////////////////////////////////////////////
// Main Function
////////////////////////////////////////////////////////////////////////////

int main(int argc,char* argv[])
{
        int i;

        ////////////////////////////////////////////////////////////////////////
        // Declare Simulation Data Storage and Status Variables
        ////////////////////////////////////////////////////////////////////////

        struct performance_data S0_data;
        struct performance_data S1_data;
        struct performance_data S2_data;
        struct sim_status temp_status;

        ////////////////////////////////////////////////////////////////////////
        // Check for proper syntax before executing simulator
        ////////////////////////////////////////////////////////////////////////

        if (argc == 3)
        {
                if (argv[1][0] == '-')
                {
                        switch(argv[1][1])
                        {
                                case 'v':
                                        if(argv[1][2] == '1') verbose = 1;
                                        else if (argv[1][2] == '2') full_debug = 1;
                                        else {printsyntax(); exit(0);}
                                        break;
                                case 'd':
                                        if(argv[1][2] == '1') debug_smalldump = 1;
                                        else if (argv[1][2] == '2') debug_fulldump = 1;
                                        else {printsyntax(); exit(0);}
                                        break;
                                default:
                                        printsyntax();
                                        exit(0);
                                        break;
                        }
                        switch(argv[1][3])
                        {
                                case 'v':
                                        if(argv[1][4] == '1') verbose = 1;
                                        else if (argv[1][4] == '2') full_debug = 1;
                                        else {printsyntax(); exit(0);}
                                        break;
                                case 'd':
                                        if(argv[1][4] == '1') debug_smalldump = 1;
                                        else if (argv[1][4] == '2') debug_fulldump = 1;
                                        else {printsyntax(); exit(0);}
                                        break;
                                case '\0':
                                        break;
                                default:
                                        printsyntax();
                                        exit(0);
                                        break;

                        }
```

```
        }
        else
        {       printsyntax();
                exit(0);
        }
}
else if (argc != 2)
{       printsyntax();
        exit(0);
}

////////////////////////////////////////////////////////////////////////
// Open the input file
////////////////////////////////////////////////////////////////////////

printf("\nTMSim 0.%f\n",VERSION_NUM);
if (argc == 2)
        read_file((char *)argv[1]);
else
        read_file((char *)argv[2]);
printf("\nStarting Simulation...\n\n");

////////////////////////////////////////////////////////////////////////
// Now simulate the input program
////////////////////////////////////////////////////////////////////////

init_pdata(&S0_data);
init_pdata(&S1_data);
init_pdata(&S2_data);

init_status("Scheme 0",&temp_status,S0_L1_SIZE,S0_L2_SIZE,0,0);
runtrace(&temp_status,&S0_data);

init_status("Scheme 1",&temp_status,S1_L1_SIZE,S1_L2_SIZE,0,1);
runtrace(&temp_status,&S1_data);

init_status("Scheme 2",&temp_status,S2_L1_SIZE,S2_L2_SIZE,1,1);
runtrace(&temp_status,&S2_data);

////////////////////////////////////////////////////////////////////////
// Simulation Complete - Calculate Average Access Times and Mem Stall Cycles
////////////////////////////////////////////////////////////////////////

printf("\nSimulation Complete.\n\n");

i=-1;
while(++i<file_data.instcount)
        S0_data.avgaccesstime += S0_data.accesstimes[i];
S0_data.avgaccesstime = S0_data.avgaccesstime / file_data.instcount;

i=-1;
while(++i<file_data.instcount)
        S1_data.avgaccesstime += S1_data.accesstimes[i];
S1_data.avgaccesstime = S1_data.avgaccesstime / file_data.instcount;

i=-1;
while(++i<file_data.instcount)
        S2_data.avgaccesstime += S2_data.accesstimes[i];
S2_data.avgaccesstime = S2_data.avgaccesstime / file_data.instcount;

////////////////////////////////////////////////////////////////////////
// Display Results
////////////////////////////////////////////////////////////////////////

printf("Instructions Traced: %d\n",file_data.instcount);
printf("Performance of Scheme 0 (No Compression):\n\n");
printresults(&S0_data);
printf("Performance of Scheme 1 (L1 Uncomp L2 Comp):\n\n");
printresults(&S1_data);
printf("Performance of Scheme 2 (Both Comp):\n\n");
printresults(&S2_data);
```

```
        ////////////////////////////////////////////////////////////////////////
        // Write Data to file
        ////////////////////////////////////////////////////////////////////////

        if(full_debug) printf("Dumping Data to File.. ");

        fprintf(file_data.datafile,"Test: %s\n",file_data.infilename);
        fprintf(file_data.datafile,"\nScheme 0,");
        dumpdata(&S0_data);
        fprintf(file_data.datafile,"\nScheme 1,");
        dumpdata(&S1_data);
        fprintf(file_data.datafile,"\nScheme 2,");
        dumpdata(&S2_data);

        if(full_debug) printf("Done.\n");

        if(full_debug) printf("Closing Data File...");
        fclose(file_data.datafile);
        if(full_debug) printf("Done.\n");

        if(debug_smalldump || debug_fulldump)
        {
                printf("Closing Debug File.. ");
                fclose(file_data.debugfile);
                printf("Done.\n\n");
        }

        if(full_debug) printf("Exiting...\n\n"); exit(0);
}

////////////////////////////////////////////////////////////////////////////
// Functions
////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////
// void runtrace(struct sim_status *,struct performance_data *)
//
// Simulates program using parameters defined during init_status
//
// receives: *current_status - data structure for status variables
//                      *pdata - structure to hold performance data
////////////////////////////////////////////////////////////////////////////

void runtrace(struct sim_status *current_status,struct performance_data *pdata)
{
        int cache_index;
        int i;

        printf("Simulating %s.. ",current_status->name);

        while(current_status->instnum++ < file_data.instcount)
        {
                ////////////////////////////////////////////////////////////////
                // Next Instruction Address is located in instruction trace array
                ////////////////////////////////////////////////////////////////

                current_status->PC = file_data.trace[current_status->instnum-1];

                ////////////////////////////////////////////////////////////////
                // Do output for each instruction
                ////////////////////////////////////////////////////////////////

                if(full_debug) printf("\n\n");
                if((verbose || full_debug) && current_status->instnum<10) printf(" ");
                if((verbose || full_debug) && current_status->instnum<100)  printf(" ");
                if((verbose || full_debug) && current_status->instnum<1000) printf(" ");
                if((verbose || full_debug)) printf("Instruction %d: PC=",current_status->instnum);
                if((verbose || full_debug) && current_status->PC<10) printf(" ");
                if((verbose || full_debug) && current_status->PC<100) printf(" ");
                if((verbose || full_debug) && current_status->PC<1000) printf(" ");
                if((verbose || full_debug)) printf("%d",file_data.trace[current_status->instnum-1]);
```

```
                if(full_debug) printf("\n");
                if(full_debug) printf("Access time is initially: %d\n",pdata->accesstimes[current_sta-
tus->instnum]);
                if(debug_smalldump || debug_fulldump) fprintf(file_data.debug-
file,"*********************************************************************\n");
                if(debug_smalldump || debug_fulldump) fprintf(file_data.debugfile,"\nInstruction %d:
PC=%d - ",current_status->instnum,file_data.trace[current_status->instnum-1]);

                ////////////////////////////////////////////////////////////////
                // Search L1 cache for instruction, if not found, update cache
                ////////////////////////////////////////////////////////////////

                cache_index = searchL1(current_status,current_status->PC);

                ////////////////////////////////////////////////////////////////
                // Test for L1 Miss - cache_index=-1 => L1 miss
                ////////////////////////////////////////////////////////////////

                if (cache_index==-1)
                {
                        ////////////////////////////////////////////////////////////////
                        // L1 Miss - Update Performance Data
                        ////////////////////////////////////////////////////////////////

                        if(full_debug) printf("  -Updating Performance Data.\n");
                        pdata->L1_misses++;

                        pdata->accesstimes[current_status->instnum] += (L1_HITTIME + L2_HITTIME);

                        if((current_status->instnum > 1000) && (current_status->instnum <= 2000))
                                pdata->L1_misses_2nd1000++;

                        ////////////////////////////////////////////////////////////////
                        // Search L2 Cache for address
                        ////////////////////////////////////////////////////////////////

                        cache_index = searchL2(current_status,current_status->PC);

                        ////////////////////////////////////////////////////////////////
                        // Test for L2 Miss - cache_index==-1 => L2 miss
                        ////////////////////////////////////////////////////////////////

                        if (cache_index==-1)
                        {
                                ////////////////////////////////////////////////////////////////
                                // L2 Miss - Update Performance Data
                                ////////////////////////////////////////////////////////////////

                                pdata->L2_misses++;
                                if(current_status->L2comp == 1)
                                        pdata->accesstimes[current_status->instnum] += L2_PENALTY_COMP;
                                else
                                        pdata->accesstimes[current_status->instnum] += L2_PENALTY_UN-
COMP;
                                if((current_status->instnum > 1000) && (current_status->instnum <=
2000))
                                        pdata->L2_misses_2nd1000++;


                                if(verbose) printf(" - L1 miss, L2 miss.\n");
                                if(debug_smalldump || debug_fulldump) fprintf(file_data.debugfile,"L1
miss, L2 miss.\n");

                                ////////////////////////////////////////////////////////////////
                                // Update L2 cache, then cache_index once L2 is current.
                                ////////////////////////////////////////////////////////////////

                                cache_index = updateL2(current_status,current_status->PC,file_data.
trace);
                        }
```

```
                        else
                        {
                                if(verbose) printf(" - L1 miss, L2 hit.\n");
                                if(debug_smalldump || debug_fulldump) fprintf(file_data.debugfile,"L1
miss, L2 hit.\n");
                        }

                        ////////////////////////////////////////////////////////////////
                        // L2 is now current, and requested address is in L2[cache_index]
                        // Now find victim frame in L1 and replace
                        ////////////////////////////////////////////////////////////////

                        cache_index = updateL1(current_status,current_status->PC,pdata);

                }
                else
                {
                        ////////////////////////////////////////////////////////////////
                        // L1 Hit
                        ////////////////////////////////////////////////////////////////

                        pdata->accesstimes[current_status->instnum] += L1_HITTIME;
                        if(current_status->L1comp == 1) pdata->accesstimes[current_status->instnum] +=
DECOMP_PENALTY;
                        if(verbose) printf(" - L1 hit.\n");
                        if(debug_smalldump || debug_fulldump) fprintf(file_data.debugfile,"L1 hit.\n");
                }
                if(debug_smalldump || debug_fulldump) fprintf(file_data.debug-
file,"*********************************************************************************\n\n");
                if(full_debug) printf("  -Access time: %d\n",pdata->accesstimes[current_status->inst-
num]);

        }
        printf("Done.\n");
}

void printsyntax()
{
        printf("\n syntax: tmsim -<options> <filename>\n");
        printf("\n options:    v1 - semi-verbose instruction trace\n");
        printf("             v2 - full debug info\n");
        printf("             d1 - dump small debug file\n");
        printf("             d2 - dump all cache info *LARGE FILE*\n\n");
        printf(" example: tmsim -v1d2 trace1.txt\n\n");
}
////////////////////////////////////////////////////////////////////////////
// void init_status(struct sim_status *,struct performance_data *,int,int,int,int,int)
//
// Initialize the status and performance_data structures for each scheme
//
// receives: *current_status - sim_status structure to be initialized
//                      size_L1 - size of L1 cache
//                      size_L2 - size of L2 cache
//                      L1comp - 1 or 0 to designated compressed cache
//                      L2comp - 1 or 0 to designated compressed cache
////////////////////////////////////////////////////////////////////////////

void init_status(char *scheme,struct sim_status *current_status,int size_L1,int size_L2,int
L1comp,int L2comp)
{
        int i;
        int j;

        if(full_debug) printf("Initializing Status Variables for %s... ",scheme);

        current_status->name = scheme;
        current_status->instnum = 0;
        current_status->PC = 0;
        current_status->L1size = size_L1;
        current_status->L2size = size_L2;
        current_status->L1comp = L1comp;
```

```
current_status->L2comp = L2comp;

        current_status->L1 = malloc(size_L1*sizeof(struct frame));
        i=-1;
        while(++i < size_L1)
        {
                current_status->L1[i].address = -1;
                current_status->L1[i].lastused = 0;
        }

        current_status->L2 = malloc(size_L2*sizeof(struct frame));
        i = -1;
        while(++i < size_L2)
        {
                current_status->L2[i].address = -1;
                current_status->L2[i].lastused = 0;
        }

        if(full_debug) printf("Done.\n");
}

////////////////////////////////////////////////////////////////////////////
// void init_pdata(struct performance_data *)
//
// Initializes cache miss counters, access time array, and average access time
//
// receives: *pdata - data structure to initiate
////////////////////////////////////////////////////////////////////////////

void init_pdata(struct performance_data *pdata)
{
        int i;

        pdata->L1_misses = 0;
        pdata->L2_misses = 0;
        pdata->L1_misses_2nd1000 = 0;
        pdata->L2_misses_2nd1000 = 0;

        if(debug_fulldump) fprintf(file_data.debugfile,"Setting up access times array:\n");
        pdata->accesstimes = malloc(sizeof(int)*file_data.instcount);
        i=-1;
        while(++i<file_data.instcount)
        {
                pdata->accesstimes[i] = 0;
                if(debug_fulldump) fprintf(file_data.debugfile,"%d  - %d",i,pdata->accesstimes[i]);
        }

        pdata->avgaccesstime = 0;
}

////////////////////////////////////////////////////////////////////////////
// void read_file(char *)
//
// Initializes the following members of file_data:
//
//              infile - input trace file from c6xsim
//              datafile - file to which to write data
//              debugfile - file for dumping cache contents
//              infilename - name of input file
//              instcount - number of instructions in trace file
//              trace - array of <instcount> instructions, read in from infile
//
// receives: *arg1 - pointer to string containing name of input file
////////////////////////////////////////////////////////////////////////////

void read_file(char* arg1)
{
        int currinst;
        ssize_t readreturn;
        size_t buflen = 0;
        char *lineptr = NULL;
```

```
char *datafilename;
        char *debugfilename;

        datafilename = malloc(sizeof(char)*(5 + strlen(arg1)));
        datafilename = strcat(datafilename,"data_");
        datafilename = strcat(datafilename,arg1);

        if(full_debug || verbose) printf("\nCreating Output Data File %s.. ",datafilename);
        file_data.datafile = fopen(datafilename,"w");
        if(full_debug || verbose) printf("Done.\n");

        ////////////////////////////////////////////////////////////////////////
        // Initialize some fields of file_data global structure
        ////////////////////////////////////////////////////////////////////////

        file_data.instcount = 0;

        if(full_debug || verbose) printf("\nOpening Input File %s.. ",arg1);
        file_data.infile = fopen(arg1,"r");
        if(full_debug || verbose) printf("Done.\n");

        file_data.infilename = arg1;

        while(readreturn = getline(&lineptr,&buflen,file_data.infile) != -1)
                file_data.instcount++;

        if(full_debug) printf("Found %d instructions, allocating memory for storage.. ",file_data.in-
stcount);
        file_data.trace = malloc(sizeof(int)*file_data.instcount);
        if(full_debug) printf("Done.\n");

        if(full_debug) printf("Reading %s.. ",arg1);

        fseek(file_data.infile,0,0);
        currinst = 0;
        while(readreturn = getline(&lineptr,&buflen,file_data.infile) != -1)
        {
                lineptr[4] = '\0';
                file_data.trace[currinst] =  strtol(lineptr,NULL, 16);
                if(full_debug && currinst < 100) printf("%d %d %d %d\n",file_data.
instcount,currinst,file_data.trace[currinst],file_data.trace[currinst]);
                currinst++;
        }

        if(debug_fulldump || debug_smalldump)
        {
                debugfilename = malloc(sizeof(char)*(6 + strlen(arg1)));
                debugfilename = strcat(debugfilename,"debug_");
                debugfilename = strcat(debugfilename,arg1);
                printf("\nCreating Output Debug File %s.. \n",debugfilename);
                file_data.debugfile = fopen("debugfile.txt","w");
        }
}

////////////////////////////////////////////////////////////////////////////
// int searchL1(struct sim_status *,int)
//
// Searches the L1 for a frame containing addr.
// Since L1 is direct-mapped, this is just a matter of checking one address.
//
// receives: *current_status - specifies which cache to search
//                      addr - address to search for in L1 cache
//
// returns: the offset of the L1 frame containing addr, if found.
//                     -1 if not found.
////////////////////////////////////////////////////////////////////////////

int searchL1(struct sim_status *current_status,int addr)
{
        int victim;
        struct frame *L1_frame;
```

```
if(full_debug) printf("  -Searching L1...");

        victim = (int)((addr / 16) % current_status->L1size);
        L1_frame = (struct frame *) &(current_status->L1[victim]);

        if(full_debug) printf(" Requested: %d found %d - ",addr,L1_frame->address);

        if(L1_frame->address == addr)
        {
                if(full_debug) printf("hit.\n");
                return(victim);
        }
        if(full_debug) printf("miss.\n");
        return(-1);
}

////////////////////////////////////////////////////////////////////////////
// int searchL2(struct sim_status *,int)
//
// Searches L2 for a frame whose address field contains addr.
// First, the set is calculated and then only that set is linearly searched.
//
// receives: *current_status - specifies which L2 cache to search.
//                      addr - address to search for in L2
//
// returns: the offset of the L2 frame containing addr, if found.
//                     -1 if not found.
////////////////////////////////////////////////////////////////////////////

int searchL2(struct sim_status *current_status,int addr)
{
        int set;
        int i;
        struct frame *L2_cache = (current_status->L2);

        set = (int)((addr / 16) % 4);

        if(full_debug) printf("  -Searching L2... Requested: %d - searching set %d.. ",addr,set);

        // Initialize loop counter i to the first index of the set
        i = set*current_status->L2size/4-1;
        // Loop from first index of set until first index of next set
        while(++i < (set + 1)*current_status->L2size / 4)
        {
                if(current_status->L2[i].address == addr)
                {
                        current_status->L2[i].lastused = current_status->instnum;
                        if(full_debug) printf(" found.\n");
                        return(i);
                }
        }
        if((full_debug)) printf(" not found.\n");
        return(-1);
}

////////////////////////////////////////////////////////////////////////////
// int updateL1(struct sim_status *,int,struct performance_data *)
//
// Updates the L1 cache.  L1 is direct-mapped.
// Here the victim frame is found and updated.
//
// receives: *current_status - sim_status structure containing L2 cache that is to be updated
//                      addr - address of requested instruction.
//                      *pdata - performance_data structure to be used in updating data
//
// returns: index of L1 into which new frame is placed
////////////////////////////////////////////////////////////////////////////

int updateL1(struct sim_status *current_status,int addr,struct performance_data *pdata)
{
        int L1_offset;
```

```
int i;
        int a_free_frame = -1;

        // Selct Victim
        L1_offset = (int)((addr / 16) % current_status->L1size);

        // Test to see if victim frame is empty
        if(current_status->L1[L1_offset].address != -1)
        {
                i=-1;
                while(++i<current_status->L1size)
                        if(current_status->L1[i].address == -1) a_free_frame = i;
        }

        if(a_free_frame != -1) L1_offset = a_free_frame;

        // Update Address field of victim frame
        current_status->L1[L1_offset].address = addr;

        // Copy contents of current_fp into data field of victim frame
        if(full_debug) printf("  -Updating L1.. Address: %d-> frame %d chosen.. ",addr,L1_offset);
        if(full_debug) printf(" Done.\n");
        if(debug_smalldump || debug_fulldump) fprintf(file_data.debugfile,"\nCache Contents After L1
Update:\n");
        if(debug_smalldump || debug_fulldump) dumpcache(current_status->L1,current_status->L1size,L1_
offset);
        if(debug_smalldump || debug_fulldump) fprintf(file_data.debugfile,"\n");

        return(L1_offset);
}

////////////////////////////////////////////////////////////////////////////
// int updateL2(struct sim_status *,int,struct performance_data *,int *)
//
// Updates the L2 cache from maim memory.  L2 is 4-way set associative with LRU
// replacement. Here, the victim frame is found and updated.
//
// receives: *current_status - status structure containing L2 cache that is to be updated
//                      addr - address of requested instruction
//                      *source - specifies which array to update L2 from - compressed or uncompressed
//
// returns: index of L2 into which new frame is placed
////////////////////////////////////////////////////////////////////////////

int updateL2(struct sim_status *current_status,int addr,int *source)
{
        // Select a victim frame
        int set = (int)((addr / 16) % 4);
        int i;
        int LRU_addr = 0;
        int a_free_frame = -1;
        if(full_debug) printf("  -Updating L2...  ");
        if(full_debug) printf("Address: %d -> Set %d",addr,set);
        i = -1;
        while(++i < (current_status->L2size/4))
        {
                if(current_status->L2[i+set*(current_status->L2size/4)].lastused <= current_status-
>L2[LRU_addr].lastused)
                        LRU_addr = i+set*(current_status->L2size/4);
                if(current_status->L2[i+set*(current_status->L2size/4)].address == -1)
                        a_free_frame = i+set*(current_status->L2size/4);
        }
        if(full_debug) printf(" frame %d chosen.. ",LRU_addr);
        if(full_debug) printf(" current address: %d ",current_status->L2[LRU_addr].address);

        if(current_status->L2[LRU_addr].address != -1 && a_free_frame != -1)
        {
                LRU_addr = a_free_frame;
                if(full_debug) printf("Frame full, free");
        }
```

```
// Replace frame
        current_status->L2[LRU_addr].address = addr;
        current_status->L2[LRU_addr].lastused = current_status->instnum;

        if(full_debug) printf("Done.\n");
        if(debug_smalldump || debug_fulldump) fprintf(file_data.debugfile,"\nCache Contents After L2
Update:\n");
        if(debug_smalldump || debug_fulldump) dumpcache(current_status->L2,current_status->L2size,LRU_
addr);
        if(debug_smalldump || debug_fulldump) fprintf(file_data.debugfile,"\n");

        return(LRU_addr);
}

////////////////////////////////////////////////////////////////////////////
// void dumpcache(struct frame *,int,int)
//
// dumpcache writes the contents of the caches to file_data.debugfile
//
// receives: *cache, a pointer to the cache to be dumped
//                      size - size of cache
//                      replaced_index - index of entry last modified
////////////////////////////////////////////////////////////////////////////

void dumpcache(struct frame *cache,int size,int replaced_index)
{
        int loop1;

        if(debug_fulldump)
        {
                fprintf(file_data.debugfile,"\nFrame Address Data\n");
                loop1 = -1;
                while(++loop1 < size)
                {
                        fprintf(file_data.debugfile,"\n ");
                        if(loop1<10)
                                fprintf(file_data.debugfile," ");
                        if(loop1<100)
                                fprintf(file_data.debugfile," ");
                        if(loop1<1000)
                                fprintf(file_data.debugfile," ");
                        fprintf(file_data.debugfile," %d  -  %d",loop1,cache[loop1].address);
                }
                fprintf(file_data.debugfile,"\n");
        }
        else if(debug_smalldump)
        {
                fprintf(file_data.debugfile,"Frame %d Modified:\nAddress\n",replaced_index);
                fprintf(file_data.debugfile," %d - ",cache[replaced_index].address);
        }
}

////////////////////////////////////////////////////////////////////////////
// void dumpdata(struct performance_data *)
//
// receives: *pdata - structure to dump
////////////////////////////////////////////////////////////////////////////

void dumpdata(struct performance_data *pdata)
{
        fprintf(file_data.datafile,"%d,",pdata->L1_misses);
        fprintf(file_data.datafile,"%d,",pdata->L2_misses);
        fprintf(file_data.datafile,"%d,",pdata->L1_misses_2nd1000);
        fprintf(file_data.datafile,"%d,",pdata->L2_misses_2nd1000);
        fprintf(file_data.datafile,"%f,",pdata->avgaccesstime);
}
```

```
////////////////////////////////////////////////////////////////////////////
// void printresults(struct performance_data *)
//
// Prints various results
//
// receives: *pdata, pointer to the performance_data structure whose data is
//                      to be displayed.
////////////////////////////////////////////////////////////////////////////

void printresults(struct performance_data *pdata)
{
       printf("   Total L1 Misses:                  %d\n",pdata->L1_misses);
       printf("   Total L2 Misses:                  %d\n",pdata->L2_misses);
       printf("   L1 Misses during 2nd 1000:        %d\n",pdata->L1_misses_2nd1000);
       printf("   L2 Misses during 2nd 1000:        %d\n",pdata->L2_misses_2nd1000);
       printf("   Average instruction access time:  %f\n",pdata->avgaccesstime);
       printf("\n");
}
```